JCL Concepts Study Guide

Interview Preparation for Mainframe Developer & Support Engineer Roles (4+ Years)

Thirteen core JCL topics — JOB/STEP/DD structure, utilities, GDGs, PROCs, SORT, COND logic, abend handling, and real production-support workflows — explained, illustrated, and paired with targeted interview questions.

About This Guide

This guide consolidates a full JCL curriculum — from basic JOB/STEP/DD structure through GDGs, PROCs, utilities, SORT, the COND parameter, and real production-support scenarios — into a single interview-preparation reference. Each topic follows a consistent Definition / Example / Concept Explanation / Flowchart / Possible Interview Questions structure, with the interview questions calibrated to what a 4+ year Mainframe Developer & Support Engineer is typically asked.

Table of Contents

1. Introduction to JCL: JOB, STEP, and DD Statements

Definition

Job Control Language (JCL) is the language used to tell the mainframe operating system (z/OS) what program(s) to run, what data those programs need, and how to allocate the system resources required to run them. A JOB is the execution of one or more programs; each individual program execution within a job is called a STEP (a single job may contain up to 255 steps); and each DD (Data Definition) statement within a step tells the system which physical dataset, device, or resource a particular file reference inside the program actually maps to.

Concept Explanation

Every JCL statement begins with // in columns 1-2. The four building blocks are the JOB statement (names the job and sets job-wide parameters), the EXEC statement (names the program or procedure a step will run), the DD statement (defines the input/output resources a step needs), and the comment statement (//* for documentation, ignored by the system). Parameters on these statements come in two flavors: positional parameters, which have no keyword and must be coded in a fixed order immediately after the statement type (e.g., the accounting information and programmer name on a JOB statement), and keyword parameters, which are named (e.g., CLASS=A) and can be coded in any order after the positional parameters. Understanding this JOB → STEP → DD hierarchy is the foundation for reading and troubleshooting any JCL.

Example

```
//JOBNAME  JOB  (098909),'TEST JOB',CLASS=A,MSGCLASS=A,
//         MSGLEVEL=(1,1),TIME=1,REGION=0M,NOTIFY=&SYSUID
//STEP1    EXEC PGM=MYPROG
//STEPLIB  DD DSN=IBMUSER.LOAD,DISP=SHR
//INFILE   DD DSN=IBMUSER.INPUT.FILE,DISP=SHR
//OUTFILE  DD DSN=IBMUSER.OUTPUT.FILE,
//            DISP=(NEW,CATLG,DELETE),
//            SPACE=(TRK,(10,5),RLSE)
//SYSOUT   DD SYSOUT=*
```

Flowchart

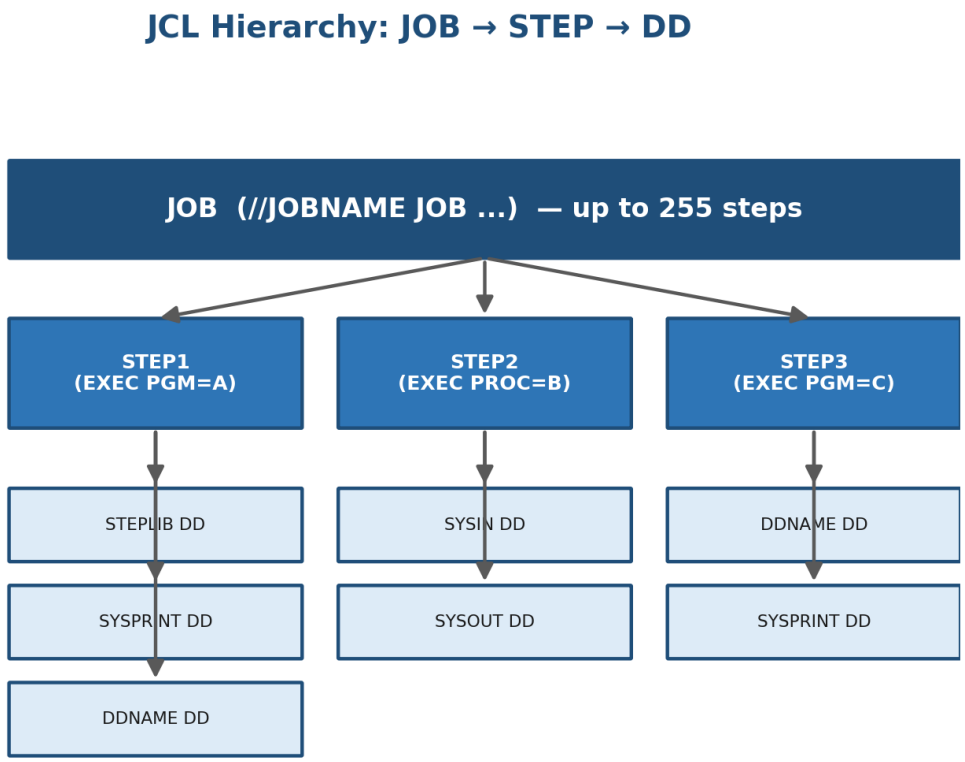

Figure 1. A JOB contains one or more STEPs; each STEP contains one or more DD statements pointing to real resources.

Possible Interview Questions

- Walk me through the structure of a JCL job you've supported — what does each part of it do?
- What is the difference between a positional parameter and a keyword parameter? Give an example of each.
- How many steps can a single JCL job contain, and what happens if that limit is exceeded?
- What is the difference between a JOB, a STEP, and a DD statement?
- How do you identify, from just looking at spool output, which step and which DD caused a job to fail?

2. JOB Statement Parameters

Definition

The JOB statement is the first statement in every job and controls job-wide behavior: how the job is tracked for accounting, which resource class it competes for, where its logs are printed, how much CPU time and memory it is allowed, whether/how to notify a user on completion, and where to restart from if rerun.

Concept Explanation

Key JOB-statement parameters include: the two positional parameters — accounting information (tracks resource usage per application) and programmer name (a short free-text description); CLASS, which buckets the job into a resource queue with a defined CPU/time/priority profile (like a movie ticket determining which screen/showtime you get); MSGCLASS, which routes the job's log output to a specific SYSOUT destination; MSGLEVEL=(statement,message), which controls how much JCL detail and how many execution messages get printed (MSGLEVEL=(1,1) is the common 'give me everything' setting, showing JOB+EXEC+DD+PROC-expanded statements plus full execution messages); TIME=(mm,ss), which caps how long the job may run before being abended with S322 (used to raise the default CLASS time limit rather than change the CLASS itself); REGION=nM, which caps how much memory the job may use (REGION=0M is a special value telling the system to allocate the maximum available); NOTIFY=&SYSUID, which sends a completion notification to the submitting TSO user (never used in production jobs — reserved for personal test jobs); and RESTART=stepname, which lets a rerun skip already-completed steps and resume from a specific step.

Example

```
//JOBNAME  JOB  (098909),'TEST JOB',CLASS=A,MSGCLASS=A,
//         MSGLEVEL=(1,1),TIME=1,REGION=0M,
//         NOTIFY=&SYSUID,
//         RESTART=STEP3
```

Possible Interview Questions

- What is the purpose of the CLASS parameter, and how does it relate to the TIME parameter when a job abends with S322?
- Why should NOTIFY=&SYSUID never be coded on a production job?
- What does MSGLEVEL=(1,1) control, and why is it the setting most support engineers prefer?
- A job abends with S322 (time exceeded). Walk me through your diagnosis and resolution using SYSMSGLG and the TIME parameter.
- How does the RESTART parameter work, and what happens to the steps before the named restart step?
- What's the difference between REGION=0M and specifying an explicit region size like REGION=4M?

3. EXEC Statement Parameters

Definition

The EXEC statement defines what a step actually executes — either a single program (PGM=) or a procedure (PROC=, or simply the procedure name by default) — and can carry step-level overrides for TIME, REGION, PARM, and COND.

Concept Explanation

PARM= is used in two distinct scenarios. First, it passes a short literal string of run-time input directly into a COBOL program (received in the LINKAGE SECTION as a length-prefixed group item, since COBOL needs to know exactly how many bytes of PARM data were supplied) — commonly used for simple developer-

supplied inputs like a country code, a Y/N flag, or a processing date. Second, PARM is used on the compiler step itself to pass compiler options that change how the load module is generated — for example, LIST (generates a COBOL listing with address/offset values used by abend-analysis tools like Abend-AID to pinpoint the failing statement), TEST (enables interactive debugging, useful for isolating the exact record that caused an S0C7), (NO)DYNAM (controls whether CALL statements resolve statically at link-edit time or dynamically at run time), and (NO)SSRANGE (enables or disables runtime subscript/index range checking — disabling it, NOSSRANGE, is why an out-of-bounds table reference surfaces as an S0C4 instead of being caught cleanly).

Example

```
//STEP1   EXEC PGM=CBLPGM01,PARM='INDIA'

* Received in the COBOL program:
LINKAGE SECTION.
01 PARM-GRP-NAME.
   03 LEN  PIC S9(04) COMP.
   03 TEXT PIC X(05).
PROCEDURE DIVISION USING PARM-GRP-NAME.
```

Possible Interview Questions

- How do you pass a run-time value into a COBOL program using JCL, and how does the program receive it?
- What's the difference between using PARM to pass business data versus using PARM to pass compiler options?
- What does the SSRANGE compiler option do, and what happens if NOSSRANGE is in effect and a program references an out-of-bounds table subscript?
- What is the difference between a static call and a dynamic call, and which compiler option controls that behavior?
- When would you use the TEST compiler option, and what production risk does leaving it on introduce?

4. DD Statement Parameters: DSN, DISP, SPACE, and DCB

Definition

The DD (Data Definition) statement is where a step's file references are connected to real, physical resources. Its core parameters are DSN (the dataset name), DISP (existence status and what to do with the dataset on normal/abnormal step completion), UNIT/VOL (physical device and volume), SPACE (how much storage to allocate), and DCB attributes such as LRECL, RECFM, and BLKSIZE (the record's physical format).

Concept Explanation

DSN can reference a permanent dataset (a normal catalogued name), a job-level temporary dataset (prefixed with &&, deleted automatically when the job ends), a step-level temporary dataset (no DSN at all, deleted when that step ends), a symbolic-parameter-driven name (prefixed with &, resolved at PROC-invocation time), or DUMMY (no real dataset — all I/O against it succeeds trivially, useful for suppressing output you don't need without changing the program). DISP is always written as DISP=(status,normal-disposition,abnormal-disposition): status is NEW (create), OLD (exclusive update), SHR (shared read), or

MOD (append); normal-disposition and abnormal-disposition are each one of CATLG (keep and catalog), KEEP (keep, uncatalogued), DELETE (discard), or PASS (hand off to a later step in the same job). SPACE=(unit,(primary,secondary),RLSE) requests storage in tracks or cylinders, with RLSE releasing any unused allocated space back to the system once the file closes; running out of allocated space produces the classic SB37/SD37 space abend. LRECL/RECFM define the record's physical shape — RECFM=FB (fixed-block) uses a constant record length for every record, while RECFM=VB (variable-block) prefixes each record with a 4-byte record-length field, saving space for data with highly variable content length at the cost of losing compatibility with some SORT processing.

Example

```
//OUTFILE  DD DSN=IBMUSER.T8.MAST.FILE,
//            DISP=(NEW,CATLG,DELETE),
//            UNIT=3390,VOL=SER=JASYS1,
//            SPACE=(TRK,(50,50),RLSE),
//            DCB=(DSORG=PS,LRECL=210,RECFM=FB,BLKSIZE=0)

//WORKFILE DD SPACE=(CYL,(10,10),RLSE)   *> no DSN, no DISP:
                                        *> defaults to (NEW,DELETE,DELETE)
```

Flowchart

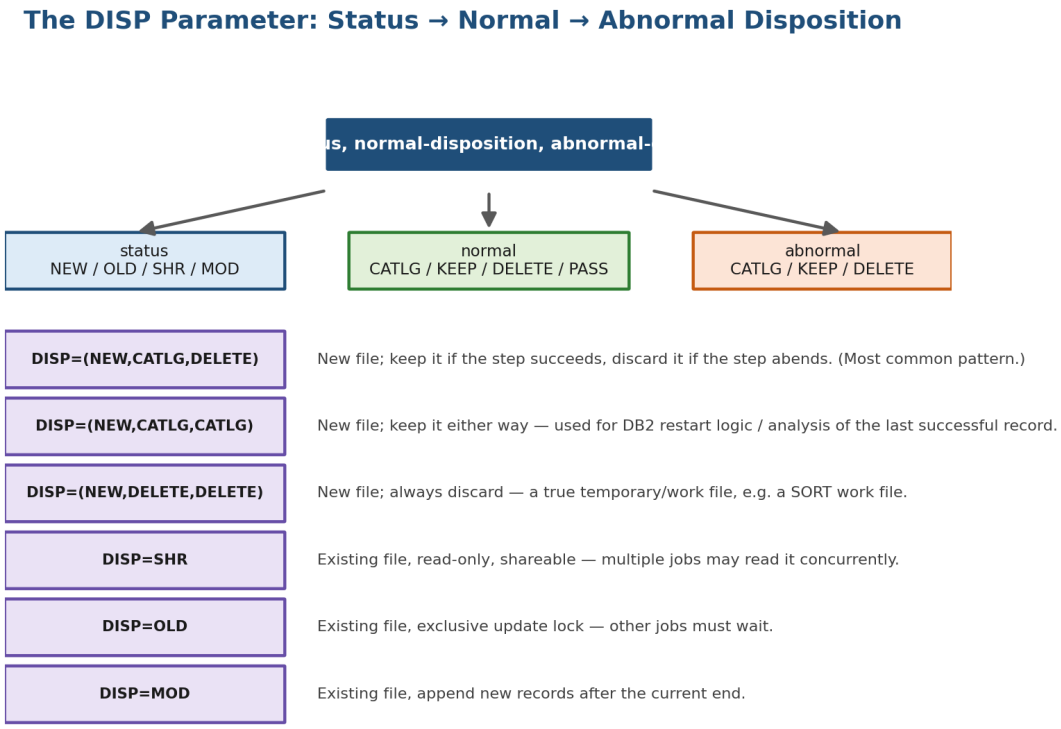
The DISP Parameter: Status → Normal → Abnormal Disposition
s, normal-disposition, abnormal-
status
NEW / OLD / SHR / MOD
normal
CATLG / KEEP / DELETE / PASS
abnormal
CATLG / KEEP / DELETE
DISP=(NEW,CATLG,DELETE)
New file; keep it if the step succeeds, discard it if the step abends. (Most common pattern.)
DISP=(NEW,CATLG,CATLG)
New file; keep it either way — used for DB2 restart logic / analysis of the last successful record.
DISP=(NEW,DELETE,DELETE)
New file; always discard — a true temporary/work file, e.g. a SORT work file.
DISP=SHR
Existing file, read-only, shareable — multiple jobs may read it concurrently.
DISP=OLD
Existing file, exclusive update lock — other jobs must wait.
DISP=MOD
Existing file, append new records after the current end.

Figure 2. DISP always resolves as status → normal-completion action → abend action; common combinations map to common real-world intents.

Possible Interview Questions

- Explain the three parts of the DISP parameter and give a real example of each combination you've used.

- What is the difference between DISP=SHR, DISP=OLD, and DISP=MOD, and when would you choose each?
- How do you resolve a space abend (SB37/SD37) in production, and what should you check before simply increasing space on one file?
- What is the difference between RECFM=FB and RECFM=VB, and what's a downside of VB files?
- What does coding a DD statement with no DSN and no DISP default to, and why is that useful for a SORT work file?
- What's the difference between a permanent dataset name, a && temporary dataset, and DISP=DUMMY?
- How do you delete an existing dataset using JCL alone?

5. Common DD Names: STEPLIB, JOBLIB, SYSIN, SYSPRINT, SYSOUT, SYSLIB

Definition

Certain DD names have special, system-recognized meaning: STEPLIB and JOBLIB tell the system where to find load modules (executable programs); SYSIN supplies input to a program or utility; SYSPRINT/SYSOUT capture diagnostic and execution output; SYSLIB supplies copybook or sub-program load locations depending on the step; SYSLMOD names the output location for a freshly link-edited load module.

Concept Explanation

STEPLIB applies to a single step only; JOBLIB (coded once, right after the JOB statement) applies to every step in the job that doesn't have its own STEPLIB. When the system needs to load a program, it searches in this order: STEPLIB (if coded for that step) → JOBLIB (if coded and no STEPLIB) → the system's default program library. If the program still isn't found, the job abends with S806 (load module not found), which is nearly always caused by an incorrect program name, an incorrect dataset name, or simply forgetting to compile/link-edit the program before running it. SYSPRINT and SYSOUT are the first two places any support engineer looks when a job fails — they contain compiler/utility diagnostic messages and program DISPLAY output respectively. SYSLIB has two different meanings depending on which step it's coded in: during a compile step, it points to the copybook library; during a link-edit step, it points to the library containing any statically-called sub-programs.

Example

```
//JOBNAME  JOB
//JOBLIB   DD DSN=IBMUSER.LOAD,DISP=SHR
//STEP1    EXEC PGM=PGMA
//STEP2    EXEC PGM=PGMB
//STEPLIB  DD DSN=IBMUSER.TESTLOAD,DISP=SHR   *> overrides JOBLIB for STEP2
only
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
  SORT FIELDS=COPY
/*
```

Possible Interview Questions

- What is the difference between STEPLIB and JOBLIB, and what is the search order the system uses to find a load module?

- What causes an S806 abend, and what are the typical root causes you've seen?
- Where do you look first when a job fails, and what kind of information do you expect to find there?
- In a compile-and-link-edit JCL, what does SYSLIB point to at each stage?
- What would you check if a job that used to run fine suddenly fails with 'load module not found' after a routine deployment?

6. JCL Utilities: IEFBR14, IEBGENER, IEBCOPY, IDCAMS

Definition

Utilities are pre-written, pre-tested, production-ready programs that perform common file-management tasks without requiring any custom COBOL code. Because a COBOL program is tightly coupled to one specific record layout, it cannot be reused across files with different layouts — utilities solve exactly this problem by working generically on any file, regardless of its internal structure.

Concept Explanation

IEFBR14 is a utility that literally does nothing except return a zero condition code; its entire purpose is to trigger the DD-statement side effects coded around it — most commonly creating an empty dataset (so a later step doesn't fail with 'dataset not found') or deleting an existing dataset via DISP=(OLD,DELETE,DELETE). IEBGENER copies data from one sequential (PS) file to another, record for record, regardless of what the records mean — it's the generic alternative to writing a COBOL read-move-write program, and avoids the file-attribute-mismatch failures (like FILE STATUS 39) that occur when a COBOL program's compiled record length doesn't match a different file's actual LRECL. IEBCOPY copies members between PDS (partitioned) datasets, commonly used to move load modules or source members between libraries. IDCAMS (Access Method Services) is the utility for VSAM and GDG management — its DEFINE command creates VSAM clusters and GDG bases, its DELETE command removes them (with FORCE to remove a GDG base and every generation in one action), and its REPRO command copies data (functionally similar to IEBGENER but VSAM-aware). In real production shops, IDCAMS DELETE is the standard way of clearing out yesterday's output files before a job recreates them today, specifically to avoid a 'duplicate dataset name' JCL failure on a daily-scheduled job.

Example

```
* IEFBR14 - delete an existing dataset
//STEP1   EXEC PGM=IEFBR14
//DDNAME  DD DSN=IBMUSER.T8.SAVE.FILE,DISP=(OLD,DELETE,DELETE)

* IEBGENER - copy PS file to PS file
//STEP1   EXEC PGM=IEBGENER
//SYSUT1  DD DSN=INPUT.FILE,DISP=SHR
//SYSUT2  DD DSN=OUTPUT.FILE,DISP=(NEW,CATLG,DELETE)
//SYSIN   DD DUMMY

* IDCAMS - delete two datasets, ignore 'not found'
//STEP1   EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN   DD *
  DELETE IBMUSER.T8.SAVE.FILE
  DELETE IBMUSER.T8.CURR.FILE
```

```
  SET MAXCC=00
/*
```

Possible Interview Questions

- Why would you use IEBGENER instead of writing a COBOL program to copy a file, especially across files with different layouts?
- What does IEFBR14 actually do, and why is it useful despite doing 'nothing'?
- A daily production job fails on its second run with a 'duplicate dataset' JCL error — what's the root cause and how do you fix it going forward?
- How would you delete an entire GDG base and all of its generations in one step?
- What does SET MAXCC=00 accomplish in an IDCAMS step, and why is it commonly coded after DELETE commands?
- What's the difference between IEBGENER and IEBCOPY?
- When would you choose SORT over IEBGENER to copy a file?

7. GDG (Generation Data Group) Concept

Definition

A Generation Data Group is a mechanism for automatically generating unique, chronologically ordered dataset names under one common 'base' name, so that a program never has to hard-code a literal file name that changes every run. Each individual dataset under a GDG base is called a generation, and is a normal sequential (PS) file whose name is system-generated.

Concept Explanation

Setting up a GDG requires two steps: first, creating the GDG base via IDCAMS DEFINE GDG, specifying the base name, a LIMIT (how many generations, 1-255, the base retains at once), an EMPTY/NOEMPTY setting (EMPTY deletes all existing generations and starts fresh once the limit is exceeded; NOEMPTY deletes only the single oldest generation, rolling the window forward by one), and a SCRATCH/NOSCRATCH setting (SCRATCH physically removes a deleted generation's data; NOSCRATCH just uncatalogs it, leaving the data recoverable). Second, generations are created and accessed using relative generation numbers: DSN=BASENAME(+1),DISP=NEW creates the next new generation in this run (only one +1 can be created per DD statement; a second new generation in the same job needs +2, and so on); DSN=BASENAME(0),DISP=SHR accesses the current/most recently catalogued generation; DSN=BASENAME(-1),DISP=SHR (and -2, -3, ...) accesses progressively older generations; and DSN=BASENAME(+1),DISP=SHR accesses a generation created earlier in the same job run, before it's been catalogued as generation 0. A critical operational detail: if a step that creates a new generation abends, that generation is never catalogued (because DISP=(NEW,CATLG,DELETE) deletes it on abend), so a rerun must go back to using +1 for that generation again rather than 0 — a common point of confusion during job restarts.

Example

```
* Create a new generation
//OUTDD  DD DSN=IBMUSER.T8.MAST(+1),DISP=(NEW,CATLG,DELETE)

* Access the current (latest) generation
```

```
//INDD   DD DSN=IBMUSER.T8.MAST(0),DISP=SHR

* Read the last 3 generations as one concatenated input
//INDD   DD DSN=IBMUSER.T8.MAST(0),DISP=SHR
//       DD DSN=IBMUSER.T8.MAST(-1),DISP=SHR
//       DD DSN=IBMUSER.T8.MAST(-2),DISP=SHR

* Read ALL generations under the base at once
//INDD   DD DSN=IBMUSER.T8.MAST,DISP=SHR
```

Flowchart

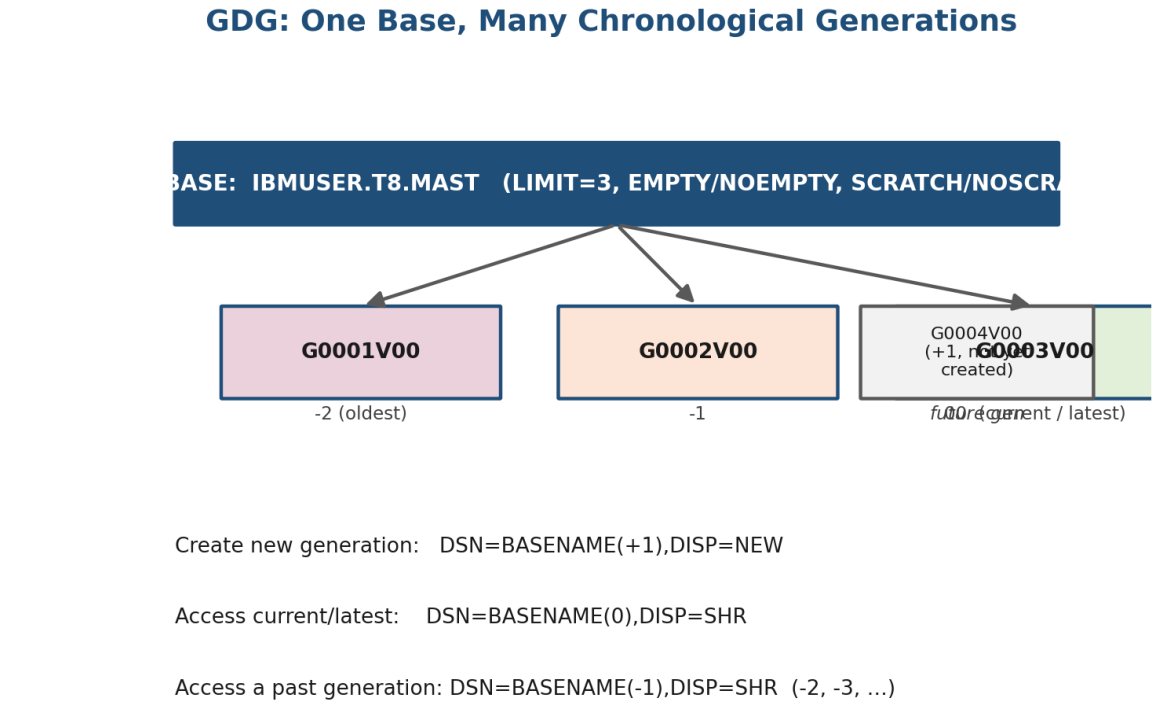

Figure 3. A GDG base owns a rolling window of generations, addressed relative to the current (0) generation.

Possible Interview Questions

- Explain how relative generation numbers work — what do +1, 0, and -1 each mean?
- What is the difference between EMPTY and NOEMPTY on a GDG base, and when would you choose each?
- A job creates a new GDG generation in Step 1 and then abends in Step 2. When you fix the issue and rerun from Step 2, what generation number do you use to read what Step 1 created, and why?
- How would a financial application retain 10 years of daily processing history using a chain of GDGs?
- What's the difference between SCRATCH and NOSCRATCH, and which would you use if you might need to recover a deleted generation?
- How do you set up a DD statement to read the last three generations of a GDG as one combined input?

- Why can only one new generation (+1) be created per DD statement, and how would you create two new generations for the same base in one job?

8. JCL Procedures (PROC): Instream vs. Catalog

Definition

A PROC is a named, reusable collection of one or more JCL steps, allowing common processing patterns (like a standard COBOL compile-and-link sequence) to be coded once and invoked by name from many jobs, rather than being duplicated everywhere. There are two kinds: instream PROCs, coded directly inside the job that uses them, and catalog PROCs, coded as a separate member in a procedure library so any job can use them.

Concept Explanation

An instream PROC is bracketed with PROC and PEND statements directly within the job stream, and can only be executed by that one job — it has no reuse value outside of that job, but is useful for one-off, job-specific step groupings. A catalog PROC is stored as its own member in a PDS (a procedure library), coded with PROC but no matching PEND, and can be invoked by any job via EXEC PROCNAME, as long as that job tells the system where to find the procedure library using a JCLLIB ORDER= statement (or, in older shops, a system-wide PROCLIB concatenation). Because the invoking job cannot code a SYSIN DD * directly for logic embedded inside a PROC, any input the PROC's steps need (SYSIN cards, control values) must instead be sourced from a control card — a PDS member specifically holding that input data, referenced with a normal DD DSN= pointing at the control card library. A very common real-world use of catalog PROCs is standardizing the COBOL compile process (precompile + compile + link-edit) once per project and requiring every application team to invoke that same shared PROC, guaranteeing consistent compiler options across the whole codebase.

Example

```
* Instream PROC (usable only by this job)
//JOB1     JOB ...
//MYPROC   PROC
//S1       EXEC PGM=PGMA
//S2       EXEC PGM=PGMB
//         PEND
//STEPX    EXEC MYPROC

* Catalog PROC (usable by any job)
* -- Member CATPROC1 in IBMUSER.T8J.PROCLIB --
//CATPROC1 PROC
//S1       EXEC PGM=PGMA
//S2       EXEC PGM=PGMB

* -- Any job invoking it --
//JOB1     JOB ...
//         JCLLIB ORDER=IBMUSER.T8J.PROCLIB
//STEPX    EXEC CATPROC1
```

Flowchart

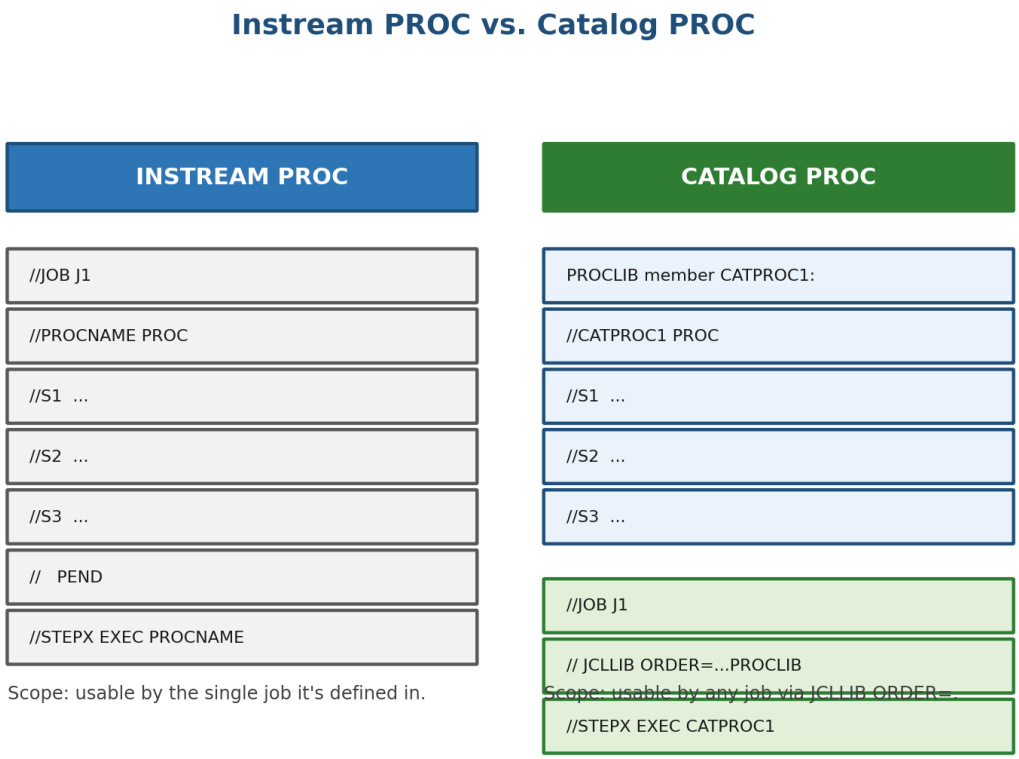

Figure 4. Instream PROCs live and die with a single job; catalog PROCs are shared library members referenced via JCLLIB.

Possible Interview Questions

- What is the difference between an instream PROC and a catalog PROC, and how does scope differ between them?
- How does a job tell the system where to find a catalog procedure library?
- Why can't you code SYSIN DD * directly for a step defined inside a PROC, and what do you use instead?
- Why would a project standardize on a shared catalog PROC for compiling COBOL programs rather than letting each developer code their own compile JCL?
- What is a control card, and how does it differ from a copybook or a load module, even though all three might live in PDS members?

9. PROC Overriding

Definition

PROC overriding is the mechanism by which a job invoking a catalog or instream PROC changes specific values inside that PROC's steps at execution time, without editing the PROC itself. This is essential in production support, where a developer or support engineer often has no direct edit access to the procedure library but still needs to adjust one value (such as allocated space) for a single run.

Concept Explanation

There are three types of PROC overriding. Symbolic parameter overriding lets a PROC define named placeholders (symbolic parameters, prefixed with &) for values the PROC's author anticipated would need to change per invocation — for example, primary/secondary space quantities or a GDG generation offset — and the invoking job supplies new values for those symbolics directly on its EXEC statement. DD-name overriding lets an invoking job replace an entire DD statement inside a specific step of the PROC, using the qualified form //stepname.ddname DD ..., commonly used to redirect a step to a different (e.g., personal, corrected) dataset for a one-time rerun. Parameter overriding works similarly for EXEC-level parameters such as PARM. A very common real-world support scenario: a job abends with a space abend (SB37) inside a step that invokes a catalog PROC whose space allocation is hard-coded as a symbolic parameter. Since the support engineer cannot edit the PROC in the production procedure library directly, they override the symbolic parameters (e.g., PQ= and SQ=, the primary and secondary space quantities) directly on the job's EXEC statement for that one rerun, without touching the PROC member at all. A durable fix (changing the PROC itself) still requires the formal change-ticket/dev/test/approval process before promoting to production.

Example

```
* PROC (in production PROCLIB) defines symbolic parameters
//CATPROC1 PROC PQ=1,SQ=1,GEN1='+1',HQ=IBMUSER
//MASTDD01 DD DSN=&HQ..T8.MAST.FILE(&GEN1),
//            DISP=(NEW,CATLG,DELETE),
//            UNIT=3390,VOL=SER=JASYS1,
//            SPACE=(TRK,(&PQ,&SQ),RLSE),
//            DCB=(DSORG=PS,LRECL=52,RECFM=FB,BLKSIZE=0)

* Job overrides PQ and SQ for this one run - no PROC edit needed
//STEPX    EXEC CATPROC1,PQ=2,SQ=2

* DD-name overriding: redirect a step's file to a personal dataset
//STEPX.MASTDD01 DD DSN=IBMUSER.PERSONAL.CORRECTED,DISP=SHR
```

Possible Interview Questions

- Describe the three types of PROC overriding and a real scenario where you'd use each.
- You get a space abend in a step that invokes a catalog PROC you don't have edit access to. Walk me through exactly how you'd resolve and rerun it.
- What is a symbolic parameter, and how does a PROC author decide which values should be exposed as symbolics?
- How do you override a specific DD statement inside a specific step of a PROC from the invoking JOB?
- What's the difference between overriding a value for a single rerun versus making a durable/permanent fix to a PROC in production?
- Explain the syntax and purpose of //stepname.ddname DD DSN=...

10. The COND Parameter: Conditional Step Execution

Definition

COND is a JCL parameter used to control whether a step executes or is bypassed based on the condition (return) codes of previously executed steps in the same job, and separately, to force a step to run (or run only) in response to an abend elsewhere in the job.

Concept Explanation

When a job step fails, it fails in one of two fundamentally different ways: an error (the step runs to completion but returns a non-zero condition code, e.g., 8 or 12, and the system by default continues to the next step), or an abend (the step is forcibly terminated by the system or the program itself, and by default the entire remaining job is stopped). For the error case, COND=(value,operator) coded on later steps tells the system to bypass that step if the comparison evaluates true against any prior step's return code (the more specific form, COND=(value,operator,stepname), ties the check to one named step only, but is used less often than the all-steps form). For example, COND=(4,LT) bypasses the step if 4 is less than any prior step's return code — so a prior return code of 0 or 4 lets the step run, while a prior return code of 8 or 12 causes it to be bypassed. For the abend case, two special COND values change behavior at the whole-job level: COND=EVEN forces a step to execute even if an earlier step abended (commonly used for cleanup or notification steps that must always run), and COND=ONLY restricts a step to execute only if an earlier step abended (commonly used for compensating/rollback logic that should never run on a normal, successful path).

Example

```
//J1       JOB ...
//S1       EXEC PGM=P1
//S2       EXEC PGM=P2,COND=(4,LT)
//S3       EXEC PGM=P3,COND=(4,LT)

* Example evaluation of COND=(4,LT) against a prior return code:
*   4 <  0  -> FALSE -> step RUNS
*   4 <  4  -> FALSE -> step RUNS
*   4 <  8  -> TRUE  -> step is BYPASSED
*   4 < 12  -> TRUE  -> step is BYPASSED

//S4       EXEC PGM=P4,COND=EVEN   *> always runs, even after an abend
//S5       EXEC PGM=P5,COND=ONLY   *> runs ONLY if a prior step abended
```

Flowchart

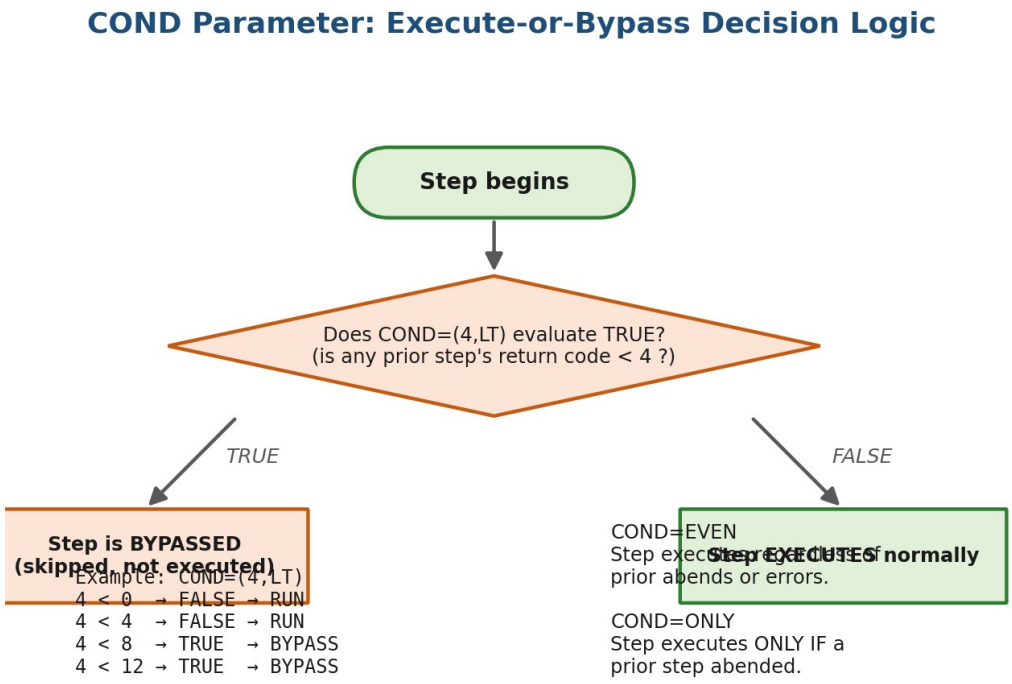

Figure 5. COND=(value,operator) bypasses a step when the comparison is true; COND=EVEN/ONLY govern behavior around abends specifically.

Possible Interview Questions

- Explain what COND=(4,LT) means and walk through how it evaluates against return codes of 0, 4, 8, and 12.
- What's the difference between how a job responds to a step ERROR versus a step ABEND by default?
- What is COND=EVEN used for, and give a real example of a step that should use it.
- What is COND=ONLY used for, and how does it differ from COND=EVEN?
- Why would you use the two-value form COND=(value,operator) instead of the three-value form COND=(value,operator,stepname)?
- You need step 5 in a 5-step job to always run for cleanup purposes regardless of what happens in steps 1-4 — how do you code that?

11. Abend Codes and Production Troubleshooting

Definition

An abend (abnormal end) is a forced termination of a step, distinct from an ordinary non-zero return code. Abends are broadly classified as system abends (raised by z/OS or a subsystem, in the SxNN family) or user abends (raised deliberately by application/COBOL logic, typically in the UNNNN family, e.g., U3000-U4000 ranges used for developer-defined business-rule failures).

Concept Explanation

Common system abends every support engineer should recognize on sight: S0C7 (data exception — invalid/non-numeric data used in arithmetic, almost always an uninitialized or corrupted numeric field); S0C4 (protection exception — an out-of-bounds memory reference, often a subscript/index error, though this can be masked if the program was compiled with NOSSRANGE); S322 (the job or step exceeded its allotted TIME); S806 (load module/program not found in the STEPLIB/JOBLIB search); S822 (insufficient REGION size); S122/S222 (job cancelled by the operator or by the submitting user, respectively — not a code failure at all); S722 (spool space exceeded, often from runaway DISPLAY/report output); and the space-abend family SB37/SD37 (a sequential/PS file ran out of allocated space) versus SE37 (a PDS ran out of directory space, generally handled differently since it involves the number of members, not raw data volume). For a space abend, the diagnostic first step is reading the abend detail line from the spool (which names the exact job, step, DD name, unit/volume, and dataset name involved), since a step with multiple output files needs to know precisely which one ran out of room — increasing space on the wrong file wastes a rerun cycle. Because space requirements tend to grow over time (e.g., a report file whose data volume increases year over year), support teams typically increase the space allocation more generously than the immediate shortfall requires, and check whether later steps or later jobs share the same growth pattern, to avoid repeatedly hitting the same abend on a predictable cadence.

Example

```
* Example space-abend detail line from spool:
B37-04,IFG0554A,IBMUSERX,
         STEP2,
         MASTDD01,
         0A82,JASYS1,
         IBMUSER.T7J.MAST.G01

* Resolution: increase space on that specific DD and rerun
//MASTDD01 DD DSN=IBMUSER.T7J.MAST.G01,DISP=(NEW,CATLG,DELETE),
//            UNIT=3390,VOL=SER=JASYS1,
//            SPACE=(CYL,(500,500),RLSE)
```

Flowchart

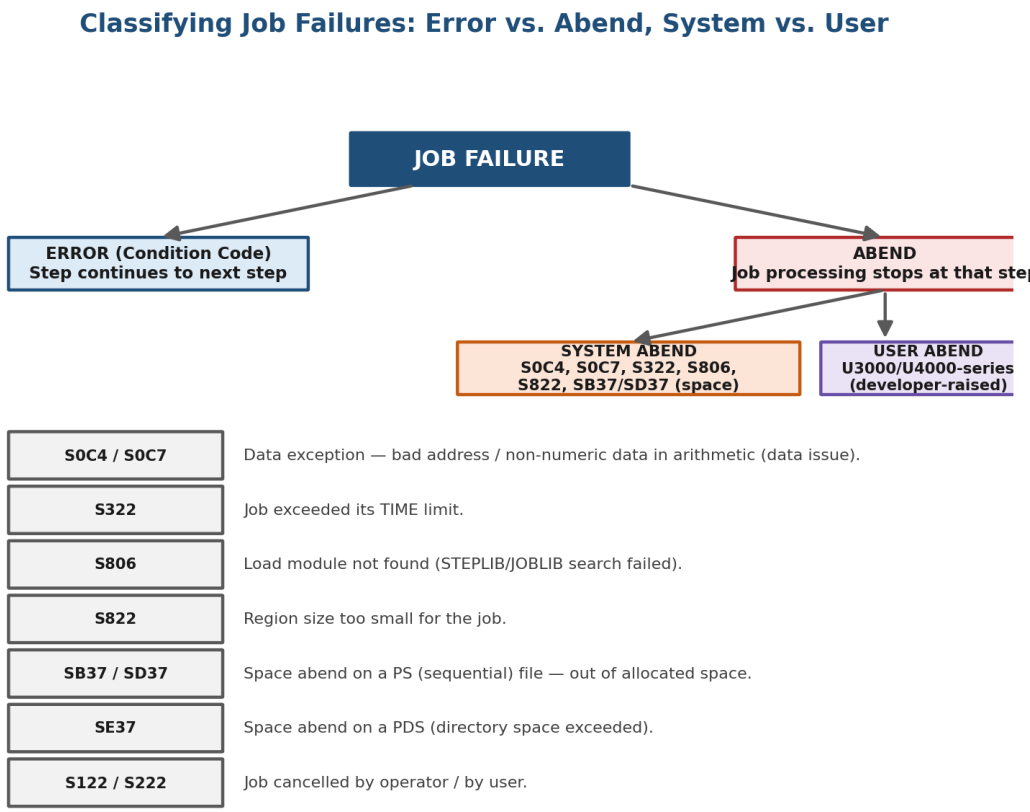
Classifying Job Failures: Error vs. Abend, System vs. User
JOB FAILURE
ERROR (Condition Code)
Step continues to next step
ABEND
Job processing stops at that step
SYSTEM ABEND
S0C4, S0C7, S322, S806,
S822, SB37/SD37 (space)
USER ABEND
U3000/U4000-series
(developer-raised)
S0C4 / S0C7
Data exception — bad address / non-numeric data in arithmetic (data issue).
S322
Job exceeded its TIME limit.
S806
Load module not found (STEPLIB/JOBLIB search failed).
S822
Region size too small for the job.
SB37 / SD37
Space abend on a PS (sequential) file — out of allocated space.
SE37
Space abend on a PDS (directory space exceeded).
S122 / S222
Job cancelled by operator / by user.

Figure 6. Failures split first into error-vs-abend, then abends split into system-vs-user, each with characteristic codes.

Possible Interview Questions

- Walk me through a real abend you resolved in production, from initial spool review through to root cause and fix.
- What is the difference between S0C4 and S0C7, and what typically causes each?
- How do you determine exactly which output file caused a space abend when a step has multiple output DD statements?
- What's the difference between a system abend and a user abend, and can you give an example of each?
- What's the difference between SB37/SD37 and SE37, and why does that distinction matter for how you fix it?
- What causes an S806 abend, and what are the most common root causes you've personally seen?
- How would you resolve an S322 abend, and what information would you gather before deciding how much to increase the TIME parameter?
- What's the difference between S122 and S222, and are either of these something you, as a developer, need to 'fix'?

12. SORT Utility: Field-Level Sorting, Filtering, and Deduplication

Definition

SORT (or DFSORT/SyncSORT, depending on the vendor) is a general-purpose utility for reordering, copying, filtering, reformatting, and deduplicating records in a file, driven entirely by control-card statements rather than custom code — making it reusable across any file layout, unlike an internal COBOL SORT verb, which is layout-dependent and requires a compiled program.

Concept Explanation

A SORT job conceptually needs three files: an unsorted input file (SORTIN), a sorted output file (SORTOUT), and one or more work files (SORTWK0n) used as scratch space while the sort algorithm rearranges data; work files are typically coded with no DSN and no DISP, which defaults to (NEW,DELETE,DELETE) — appropriately temporary. The core control statement, SORT FIELDS=(start,length,format,order), identifies which bytes to sort on, their data format, and ascending (A) or descending (D) order; the format code must match how the field is actually stored — CH for character/display data, ZD for zoned decimal, PD for packed decimal (COMP-3), and BI for binary (COMP). Beyond straightforward sorting, SORT supports several very common production patterns: SORT FIELDS=COPY passes every record through unchanged (useful purely for copying/concatenating files, especially large or tape-resident files where IEBGENER may be less efficient); INCLUDE COND=/OMIT COND= filter records based on field values (functionally similar to a SQL WHERE clause); INREC=/OUTREC= reformat, reorder, or trim columns on the way in or out; and pairing SORT FIELDS=(key) with SUM FIELDS=NONE removes duplicate records that share the same sort key, keeping only one occurrence of each — a very common way to deduplicate a file without writing custom COBOL logic.

Example

```
//STEP1    EXEC PGM=SORT
//SYSPRINT DD SYSOUT=*
//SORTIN   DD DSN=INPUT.FILE,DISP=SHR
//SORTOUT  DD DSN=OUTPUT.FILE,DISP=(NEW,CATLG,DELETE)
//SORTWK01 DD SPACE=(CYL,(10,10),RLSE)
//SYSIN    DD *
  SORT FIELDS=(1,5,CH,A)
/*

* Field format examples for an account file:
* ACCT-NUM  PIC X(05)          -> SORT FIELDS=(1,5,CH,A)
* ACCT-BAL  PIC 9(06)          -> SORT FIELDS=(6,6,ZD,A)
* ACCT-BAL  PIC 9(06) COMP     -> SORT FIELDS=(6,4,BI,A)
* ACCT-BAL  PIC 9(06) COMP-3   -> SORT FIELDS=(6,4,PD,D)

* Remove duplicate records sharing the same key
//SYSIN    DD *
  SORT FIELDS=(1,5,CH,A)
  SUM FIELDS=NONE
/*
```

Flowchart

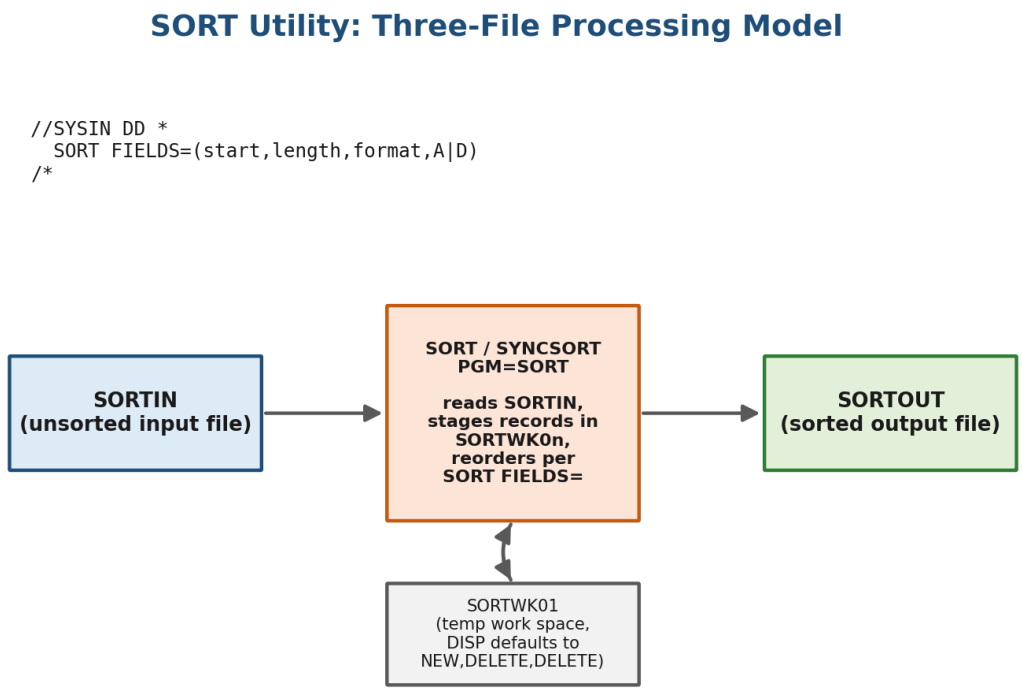

Figure 7. SORT reads SORTIN, stages data in SORTWK0n, and writes the reordered result to SORTOUT, driven by control-card logic.

Possible Interview Questions

- Why would you use the SORT utility instead of the COBOL SORT verb to reorder a file?
- What is the difference between internal sort (COBOL) and external sort (utility), and what are the tradeoffs of each?
- Explain the four parts of a SORT FIELDS statement and give an example for a packed-decimal field.
- How would you remove duplicate records from a file using SORT, keeping only one occurrence of each key?
- What is SORT FIELDS=COPY used for, and when would you prefer it over IEBGENER?
- What's the difference between INCLUDE COND and OMIT COND?
- What is the purpose of the SORTWK0n DD statement, and what does its default disposition tell you about its intended lifespan?
- How would you sort on a COMP-3 field, and what format code does that require?

13. Production Support: Working Without Direct PROD Access

Definition

In most mainframe shops, support engineers and developers do not have direct edit access to production JOB libraries, PROC libraries, or datasets. Instead, changes for a rerun are made through a scheduler tool's

copy of the job, through PROC overriding, or through personal copies of datasets — with any permanent fix to production routed through a formal change-ticket process.

Concept Explanation

This constraint shapes the entire day-to-day support workflow. If a job (not a PROC) needs a one-time change to rerun, the engineer copies the job from the scheduler tool's production library into a personal/working copy, edits it within the scheduler tool itself, and resubmits from there — editing a scheduler's local copy without resubmitting through the tool has no effect on the actual production run (the job will simply abend again on its next scheduled trigger). If the change needed is inside a PROC (which the scheduler typically doesn't copy, since it only copies the job's own lines), PROC overriding (symbolic parameter, DD-name, or EXEC-parameter overriding, coded directly in the job) is used instead. If the issue is bad or incomplete data in a production dataset that needs correcting before a rerun, the standard sequence is: copy the production dataset to a personal dataset (using 3.4's copy option, IEBGENER, IDCAMS REPRO, SORT+COPY for very large or tape files, or a file-editing tool), edit the personal copy (ISPF browse/edit for huge files, since a full edit session can be too heavy for million-record files; a dedicated file tool — one that understands the copybook layout — for smaller, more surgical edits), optionally compare the corrected personal copy against the original production file (using ISPF option 3.13/SUPERC, the standard tool for any file or code comparison, or IEBCOMPR), and finally point the job at the corrected personal file for the rerun — either by editing the DSN directly in the job/scheduler copy, or via DD-name overriding at the JOB-PROC-STEP level. Any of these workarounds are rerun-only fixes; a durable, permanent change to the production JOB, PROC, or reference data always requires a formal change ticket, development-environment testing, and management/change-board approval before being applied to the actual production library.

Example

```
* Compare a corrected personal file against the original production file
* using ISPF option 3.13 (SUPERC)

* DD-name overriding to point a rerun at the corrected personal file,
* qualified down to the exact step inside a PROC:
//STEPX.PROCSTEP1.MASTDD01 DD DSN=IBMUSER.PERSONAL.CORRECTED,DISP=SHR

* Copying a production file to a personal dataset via IDCAMS REPRO
//STEP1    EXEC PGM=IDCAMS
//INPUTDD1 DD DSN=PROD.FILE,DISP=SHR
//OUTPUTD1 DD DSN=IBMUSER.PERSONAL.COPY,DISP=(NEW,CATLG,DELETE)
//SYSIN    DD *
  REPRO INDD(INPUTDD1) OUTDD(OUTPUTD1)
/*
```

Flowchart

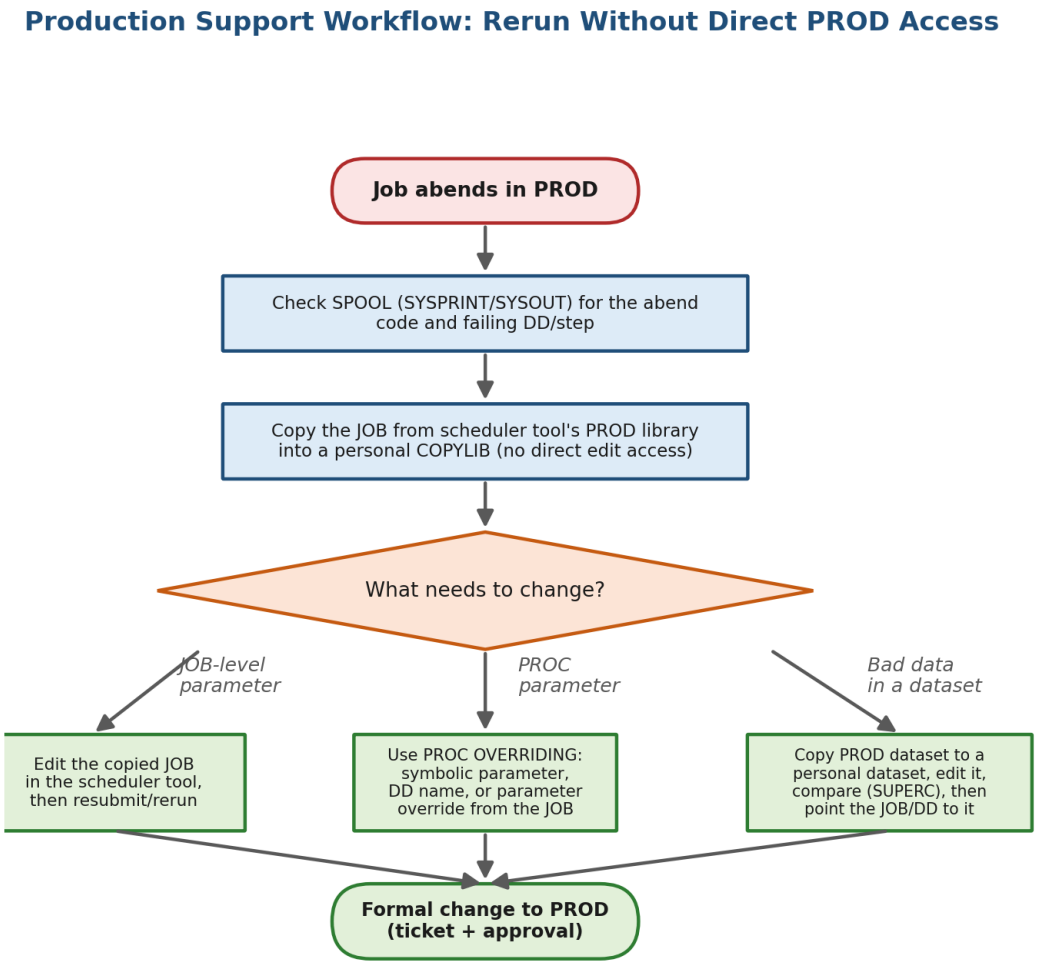
Production Support Workflow: Rerun Without Direct PROD Access
Job abends in PROD
Check SPOOL (SYSPRINT/SYSOUT) for the abend code and failing DD/step
Copy the JOB from scheduler tool's PROD library into a personal COPYLIB (no direct edit access)
What needs to change?
JOB-level parameter
PROC parameter
Bad data in a dataset
Edit the copied JOB in the scheduler tool, then resubmit/rerun
Use PROC OVERRIDING: symbolic parameter, DD name, or parameter override from the JOB
Copy PROD dataset to a personal dataset, edit it, compare (SUPERC), then point the JOB/DD to it
Formal change to PROD (ticket + approval)

Figure 8. Three different problem types — job-level, PROC-level, and data-level — route through three different rerun mechanisms, all converging on a formal change process for a permanent fix.

Possible Interview Questions

- Walk me through your process for rerunning a production job when you don't have direct edit access to the production JOB library.
- Why does editing a job copy in the scheduler tool without resubmitting from that tool not actually fix the production job?
- How do you correct bad data in a production dataset when you don't have edit access to production datasets directly?
- What tool do you use to compare two versions of a file or program, and what is it commonly used for beyond just data comparison?
- Explain DD-name overriding at the JOB-PROC-STEP level and give a real scenario where you'd use it.
- What is the difference between a rerun-only workaround and a permanent production fix, and what process governs the latter?

- If a job's fix requires a change to a PROC rather than the job itself, why can't the scheduler tool's normal 'copy and edit' process help you, and what do you use instead?